Description:

In this project, we are expected to implement an interpreter for an assembly language of a CPU called HYP86*. We are given an input file that contains a code segment in assembly language and we process this code segment with the interpreter that we implement in Java or C++ . While processing the code segment , we need to set the flags , registers , 64 Kb memory correctly and print the output of assembly code to console .

(*There are some assumptions about instructions , flags and variables . For example , we are are expected to implement only ZF,CF,AF,SF,OF flags.)

Project Details:

We implemented this project in groups of 2 students and used C++ language . At the beginning of the project , we decided to structure of the project together (for example , how many function we need to implement , what parameters that each functions take and what they need to return) and shared the tasks . We implemented inside of the functions separately but always we were in contact and exchanged ideas.

Structure Of Project :

(Important note : We are taking input as an argument and print output to the console)

(Our project is compling and can handle all test cases)

Important global variables :

1. "memory" array : each index corresponds to 1 byte and size of this array is 65536.

2."labels" vector consists of < string , unsigned int > pairs. Strings refer the label name , unsigned int refers the starting index of this label in memory .

3."reg" vector consists of < string , unsigned int > pairs. Strings refer the register names and unsigned int refers value of this register.

4."code" vector contains tokens of the input file.

(*We are using unsigned int because there are no negative integers in our project)

Using the reg vector :

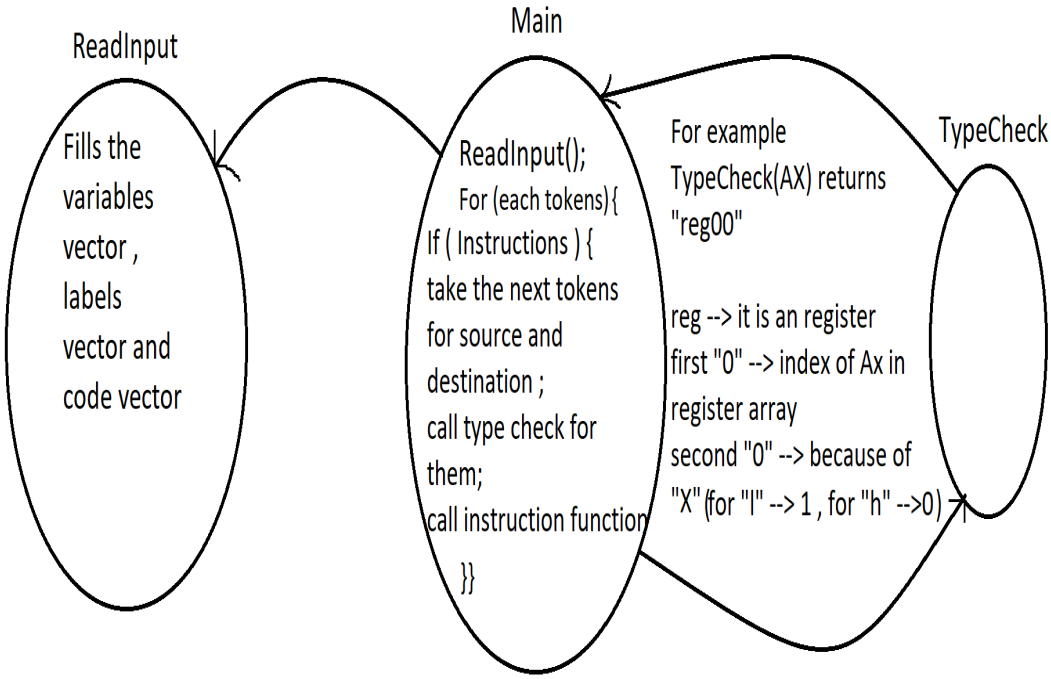
This vector contains only 16 bit registers . Therefore, when we try to access to low part or high part of the 16 bit registers , we just get the remainder or quotient of dividing by 256.

Using of labels vector :

If we encounter with an label during processing of assembly code , we search for it in our labels vector. If there is a label in this vector with the same name , we get it's unsigned int pair and jump this label. If there is not , we just give error . (Because at the beginning , we scan all code for one time and get all the labels and variables .)

Main , ReadInput and TypeCheck functions :

At the beginning of the main, readInput function is called . This function parses the input file , inserts the tokens to "code" vector , fills "labels" and "variables" vectors , allocate space for each instructions and variables (6 byte for each instructions) . (ReadInput function can handle various type of inputs . Such as (Mov Ax, Bx) or (ADD [1234h] ,ax)). After this process , In main , tokens in "code" vector are sending to the TypeCheck function . This function detects the features of tokens such as register ,memory location, word size ,byte size . Then it returns parameters that will be given to instruction functions by concataneting necessary informations as a string . In main this strings are splited and cast to unsigned integer . If there is a error in this informations , main prints error and terminates . If there is not , it gives the unsigned integers to instruction functions as parameters .



Helper functions :

There are some helper functions in our project that are used to provide some functionalities . They have clear names and it is easy to understand what these functions do with their name. For example , hex2dec function takes a string of a hexadecimal number and returns in decimal form .

Instruction functions :

These functions are generally called in main with the parameters that are taken from TypeCheck function . They sets flags , registers and memory according to their inputs . Most of them have same names with the instructions in assembly language .

Arguments of instuction functions :

InsFunc(unsigned int loc1, unsigned int loc2, int type, int type2, int type3 , int type4) {

}

possible arguments that an instruction functions can take are the arguments above and all of the instruction functions take various combinations of these arguments. (except rcr2,rcl2,shl2,shr2)

loc1----> all instruction functions take loc1 as an argument . It refers the location of our destination element in reg vector or memory array . (For MOV AX,BX , loc1 is 0 because index of AX in reg vector is 0 . For ADD [1234d],BX , loc1 is 1234 which refers the index in memory array).

loc2-----> It works the same as loc1 but it refers to location of source element . Some instruction functions do not take it as an argument (for example mul function because it has a syntax like MUL BX and BX is being sent by loc1 .)

type ----> loc1 and loc2 may be an index of a register , memory location or constant . So we need to give an argument refers which combination of loc1 and loc2 is sent to instruction function . For example, for add function :

add <reg>,<reg>

add <reg>,<mem>

add <mem>,<reg>

add <reg>,<con>

add <mem>,<con>

there are this kind of combinations. type==1 refers <reg> <reg> , type==2 refers <reg><mem> , etc. (these information writes on the top of the functions as a comment).

type2----> When loc1 is location of register , it indicates whether loc1 refers 16 bit register or low register or high register .(type2==0 means "X" , ==1 means "H" , ==2 means "L") . For example , for MOV AL ,BL , type2 is 2 .

When loc1 is location of memory , it indicates if loc1 refers word size memory or byte size memory location . (type2==0 means word size (w[1234h]) , type2==1 means byte size (b[1234h]) , type2==2 refers ([1234h])).

type3----> It works same as type2 but it is for loc2 .

type4----> logicalEx ,ShiftAndRotate, jmp and IncAndDec functions handle more than 1 operation . For example ShiftAndRotate function handles SHL,SHR,RCL,RCR operations . Type4 refers for which operation these functions are called .

(***Important Note : Arguments can move in some functions. For example , If there is not loc2, type3 behave as type4 and type4 are not sent .)

(***Meaning of arguments are clearly explained on top of the functions in the code)

Small notes on Instruction functions :

1. These functions are generally contains the big if statements for each value of "type" argument.

2. They equal flags that this operation affects to zero at the beginning of the function and make them one when necessary conditions are met .

3.Syntactic errors are checking in main and typeCheck functions but logical error are checking in the operation functions and if there is an condition that is not allowed these functions returns 0 else they return 1. For example if you try to mov AL,BX which means you are trying to move a 16 bit register to 8 bit register , mov function return 0 .

IncAndDec and logicalEx functions :

These functions do not have same name with the operations they do . IncAndDec function does the increment and decrement operations , logicalEx function does all logical expression according to given argument .

ShiftAndRotate , rcr2, rcl2, shl2, shr2 functions :

These functions are connected to the ShiftAndRotate function . ShiftAndRotate gives them 2 unsigned int. One of them is number that will be shifted or rotated and other one is counter which shows how many times shifting or rotating operation should be done . rcr2,rcl2,shl2,shr2 recursively call itself .

Compare instruction

This instruction does not have any spesific function. For this instruction we just call sub function with type4==1 condition. When the sub function is called as such , it does not update anything but flags.

Challenges During Implementation :

We think one of the hardest part of the this task was parsing given assembly code because there are too many different syntax type for each instruction that assembly does not give error. For example, there may be space between source and destination register or not . (Mov AX,BX or Mov AX , BX) . Another tough challenge was finding out when the flags are setting . Flags may behave differently for some operations and sometimes we had difficulty to setting flags correctly.